Implementation, implementation, implementation: old and new options for putting surveys and experiments online

George MacKerron
Department of Geography & Environment
London School of Economics & Political Science

18 April, 2011

Abstract

The Internet offers enormous possibilities for surveys and experimental data collection, including randomised treatments, customisation, and interactivity. These capabilities are well suited to the implementation of choice modelling experiments.

However, the implementation of web surveys is not a simple task, and the existing options open to researchers are commonly unsatisfactory in a number of ways. The result is that few Internet surveys and experiments are able to exploit the unique capabilities of the web.

This paper suggests a new approach, illustrated with a working prototype: an open-source, domain-specific language (DSL) designed for specifying web surveys and experiments, which is called *websperiment*.

The paper first looks at the existing approaches, highlighting their strengths and weaknesses. It then outlines the concepts underlying *websperiment*, and this DSL's nature and scope, with simple code examples. Finally, it shows how the DSL can be used to concisely specify a highly dynamic choice modelling survey.

1 Introduction

The Internet offers enormous possibilities for survey and experimental data collection: randomisation, customisation, interactivity, paradata[1], and more. Web surveying is cheaper and faster than traditional approaches, and as universal Internet access edges ever closer, its biggest single drawback—an incomplete and biased sampling frame—is set to keep on diminishing.

There has, rightly, been much interest in mode effects and in the sampling, validity, and design issues associated with web surveys (e.g. Marta-Pedroso et al., 2007; Dillman et al., 1998; Schonlau et al., 2002; Couper, 2008). There has, however, been rather little discussion regarding the practical implementation of such surveys.

Survey implementation matters, and it matters arguably even more on the web than in some other modes: an online instrument must compensate for the lack of trained interviewers to administer it. Web survey implementation affects accessibility, compatibility and consistency across respondents; it affects respondent motivation and experience; it creates context effects, and has implications for data security.

[1]Paradata are data that "do not describe the respondent's answers but the process of answering the web questionnaire" (Heerwegh, 2003), for example, the time a respondent spends answering a specific question, or the sequence in which response options are chosen.

Implementation is a genuinely difficult problem. It is subject to the combined challenges both of good generic survey implementation and of good generic web application implementation[2]. It is a problem which increases in scale in tandem with the potential rewards: "the more complexity one builds into the instrument, the greater the cost and effort required to program and test the survey, and the greater the likelihood that something might not work as intended" Couper (2008, 30).

Implementation is arguably a problem which has yet to be satisfactorily addressed. Few Internet surveys currently exploit the unique capabilities of the web: most function simply as on-screen representations of a paper-and-pencil design. Furthermore, even such unambitious representations are rarely executed well. Couper (2008, xvi) appears justified in his continuing amazement "at the poor design of many web surveys".

In the hope of helping to improve web survey implementation, this paper suggests a new approach, embodied in and illustrated by a working prototype. The approach has three distinctive aspects. First, it involves the development of a 'domain-specific' programming language (DSL). Second, it makes liberal use of a mechanism known as 'inheritance', an important element in object-oriented programming. Finally, it is open source. These are somewhat technical concepts, and *prior knowledge is not assumed*. They are considered in more detail in section 3.

The rest of the paper is structured as follows: section 2 looks at the main options currently available to researchers implementing web surveys, and their advantages and limitations; section 3 describes the DSL approach; section 4 illustrates the potential for its application to a web-based choice experiment; and section 5 concludes.

2 Existing survey implementation options

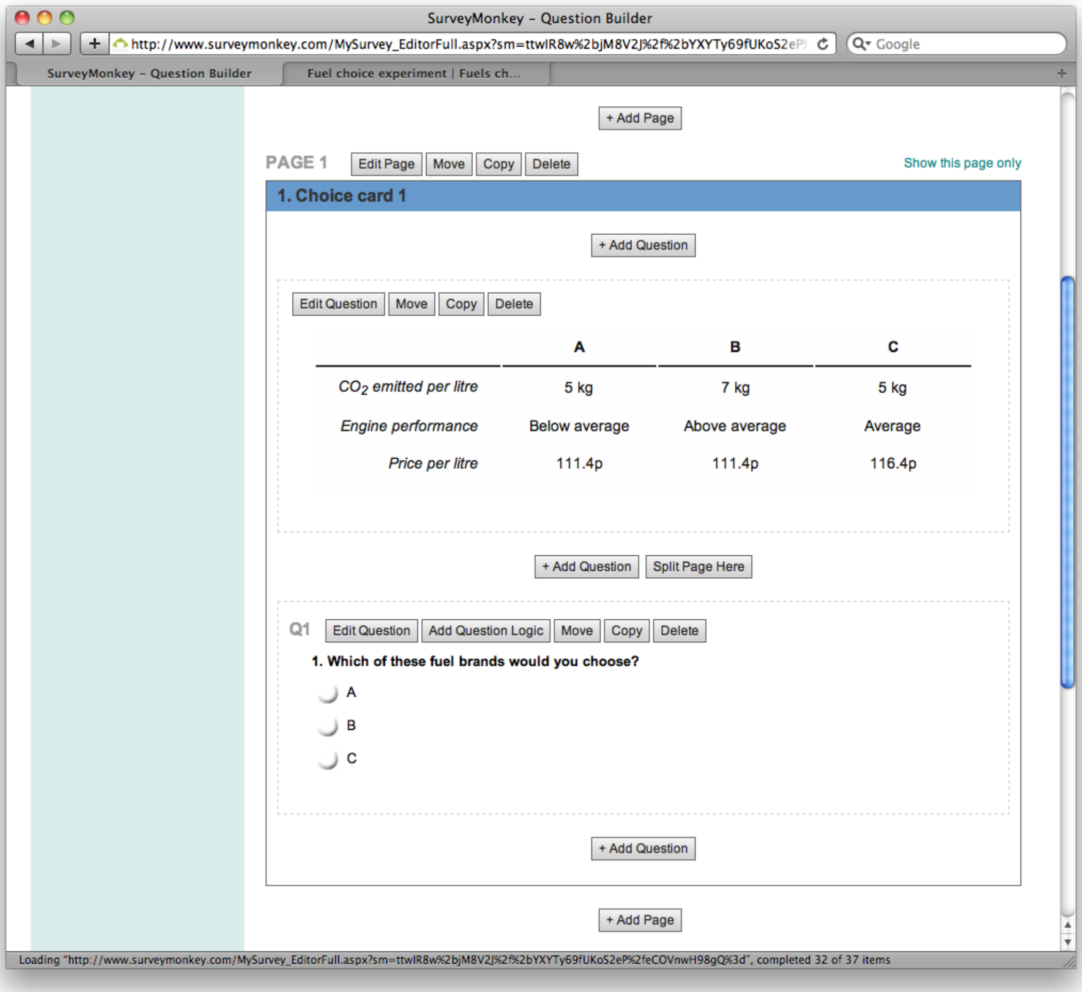
Researchers trying to implement a survey or experiment online generally choose one of four major options: a managed web-based service; a locally-installed software application; a specialist consultancy; or some form of Do-It-Yourself (that is: pick a programming language, and start writing code).

(a) Managed web-based service

These services enable a survey to be designed via a web-based Graphical User Interface (GUI), and then hosted on the provider's server. Popular web-based services include: SurveyMonkey, QuestionPro, Zoomerang and Wufoo, which are principally business-focused; Bristol Online Surveys, which has a more academic flavour; and various offerings from Confirmit, which aims more specifically at market research professionals. Typically a web-based service is a relatively low-cost option offering rather basic features. Often it also produces an unattractive and poorly accessible survey (Kuipers, 2005).[3]

[2]The latter include not only the general challenges associated with IT service development, in which effective communication of requirements is a key factor, but a further host of technical characteristics related to the web. For example: respondents must be tracked across separate page requests, overcoming the statelessness of the underlying protocol, HTTP; limited, inconsistent and incompatible browser, operating system and hardware capabilities and settings must all be accommodated; logic must be split or duplicated between the server and untrusted clients; and in spite of all this, an attractive, consistent and easily navigated user interface must be maintained.

[3]For example, the popular service Survey Monkey uses non-standard custom form controls (check-boxes, radio buttons and so on). These controls are unfamiliar to web users, and raise accessibility issues both in terms of disabled users (screen reading software will not recognise them) and web browser configuration (respondents without JavaScript enabled, including many who are subject to institutional IT security policies, see only the message "JavaScript is required for this site to function, please enable"). Bristol Online Surveys, to which many UK academic institutions subscribe, provides no routing capabilities and does not use the HTML `<label>` tag for form controls (this is disability-unfriendly, and also makes the controls an inconveniently small click target). Some other



	A	B	C
CO_2 emitted per litre	5 kg	7 kg	5 kg
Engine performance	Below average	Above average	Average
Price per litre	111.4p	111.4p	116.4p

1. Which of these fuel brands would you choose?

- A
- B
- C

Loading "http://www.surveymonkey.com/MySurvey_EditorFull.aspx?sm=ttwIR8w%2bjM8V2J%2f%2bYXYTy69fUKoS2eP%2feCOVnwH98gQ%3d", completed 32 of 37 items

Figure 1: Creating a basic choice card, using an image for the choice presentation, using SurveyMonkey.

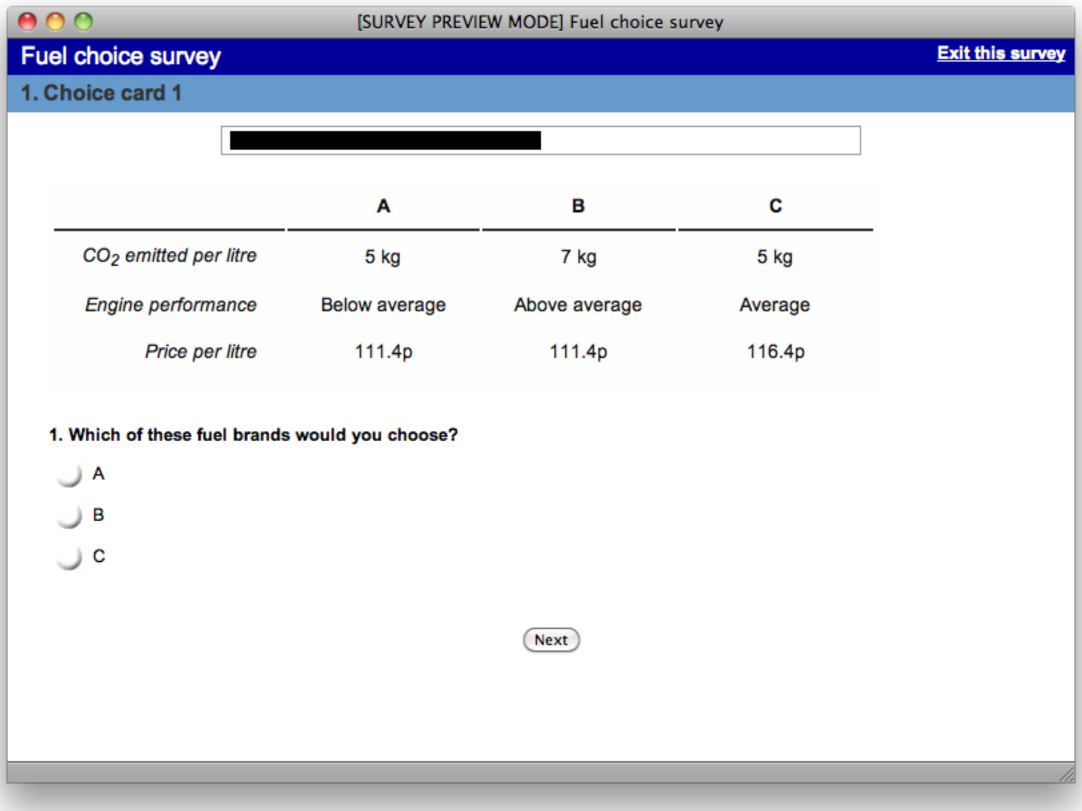
It is possible to implement a choice experiment with these services via the traditional pen-and-paper strategy of creating multiple versions of the survey instrument, differing only in the attribute levels displayed within the choice cards section (this can be coupled with a simple server-side script which redirects visitors to one of the multiple survey versions at random). Depending on the capabilities of the service used, formatted items such as tables may have to be inserted as images, and in this case the information will not be accessible to screen-reader software.

Figures 1 and 2 illustrate the use of SurveyMonkey to design a choice card, and the resulting survey display.

(b) Locally-installed software application

The boundaries between web-based services and locally-installed software are increasingly becoming blurred, but locally-installed software generally provides a more advanced feature set then the web-based services discussed above. Like web-based services, locally-installed software generally offers a comprehensive GUI for survey design. GUIs should facilitiate discovery of the software's features and eliminate some kinds of input error. Survey design

services provide no 'previous page' control (and break when the browser's 'back' button is used), require Java or Adobe Flash, or exclude non-Windows users.


[SURVEY PREVIEW MODE] Fuel choice survey

Fuel choice survey

Exit this survey

1. Choice card 1

	A	B	C
CO_2 emitted per litre	5 kg	7 kg	5 kg
Engine performance	Below average	Above average	Average
Price per litre	111.4p	111.4p	116.4p

1. Which of these fuel brands would you choose?

- A
- B
- C

Next

Figure 2: Choice card page designed and hosted using SurveyMonkey.

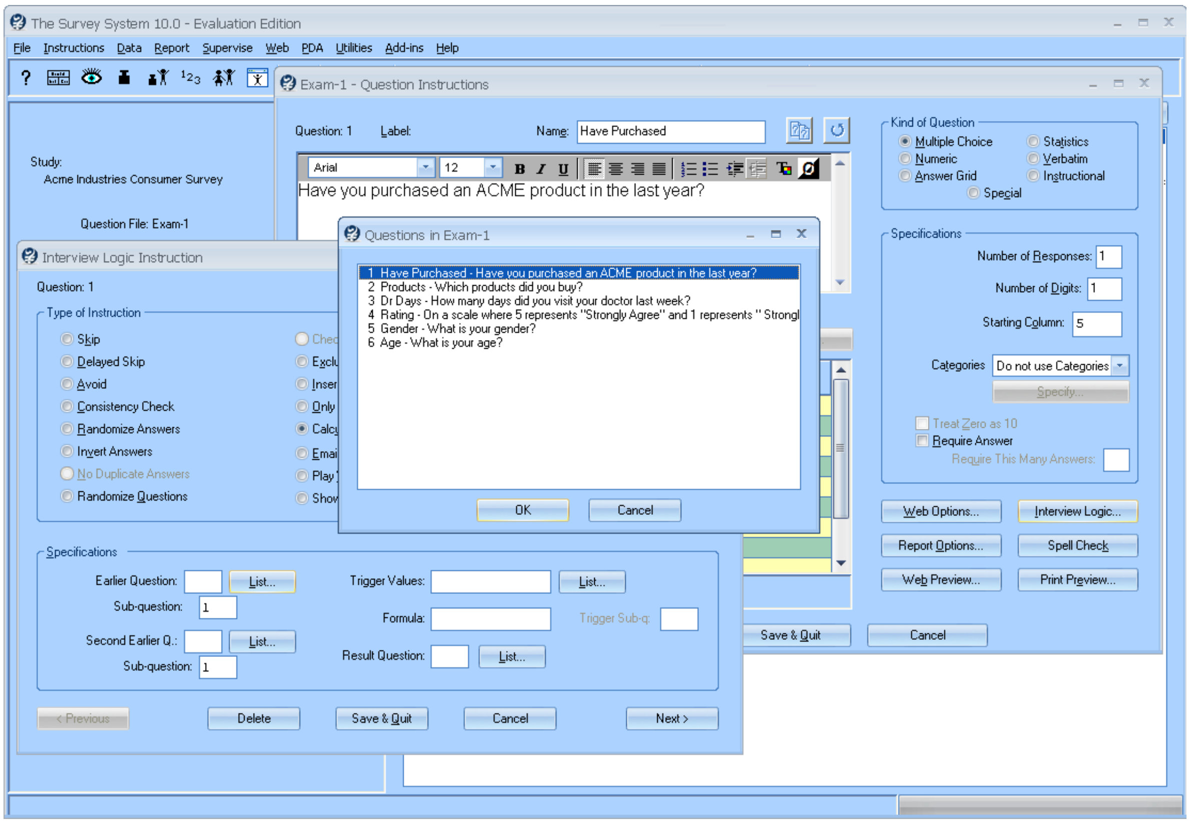
GUIs may be somewhat clumsy, however, with option-heavy dialogue-boxes nested many levels deep (see The Survey System software illustrated in Figure 3, for example).

Locally-installed survey software may be expensive, particularly if the cost cannot be spread across multiple projects, and is generally subject to restrictive licencing terms regarding who may install and use it, on what scale, and for how long. Once a survey is designed, arrangements must then be made for hosting online. Most software providers offer paid hosting options, and/or enable the export of scripting files and supporting resources for hosting on your own server. The latter option will generally require particular server features, such as the availability of a scripting language (e.g. Perl) or the use of a particular platform (e.g. Microsoft Windows with Internet Information Services (IIS) server software).

Some software is designed specifically for the implementation of online choice experiments, and can automatically generate a range of experimental designs based on the attributes and levels specified—an extremely useful capability. Figures 4 and 5 and show the design and display of a choice experiment using one such software package, SSI Web.

(c) Specialist consultancy

The results of engaging consultants ultimately depend, obviously, on the skills and experience of the consultants. However, a consultancy itself may well be using one of the options discussed above, in which case the service offered will be subject to the same strengths and limitations. Achieving precise communication of requirements is likely to be a significant challenge, and some level of control will be lost. This option is also liable to be expensive.



Study: Acme Industries Consumer Survey

Question File: Exam-1

Question: 1 Label: Name: Have Purchased

Have you purchased an ACME product in the last year?

Kind of Question: [✓ Multiple Choice] Numeric Answer Grid Statistics Verbatim Instructional Special

Specifications: Number of Responses: 1; Number of Digits: 1; Starting Column: 5; Categories: Do not use Categories

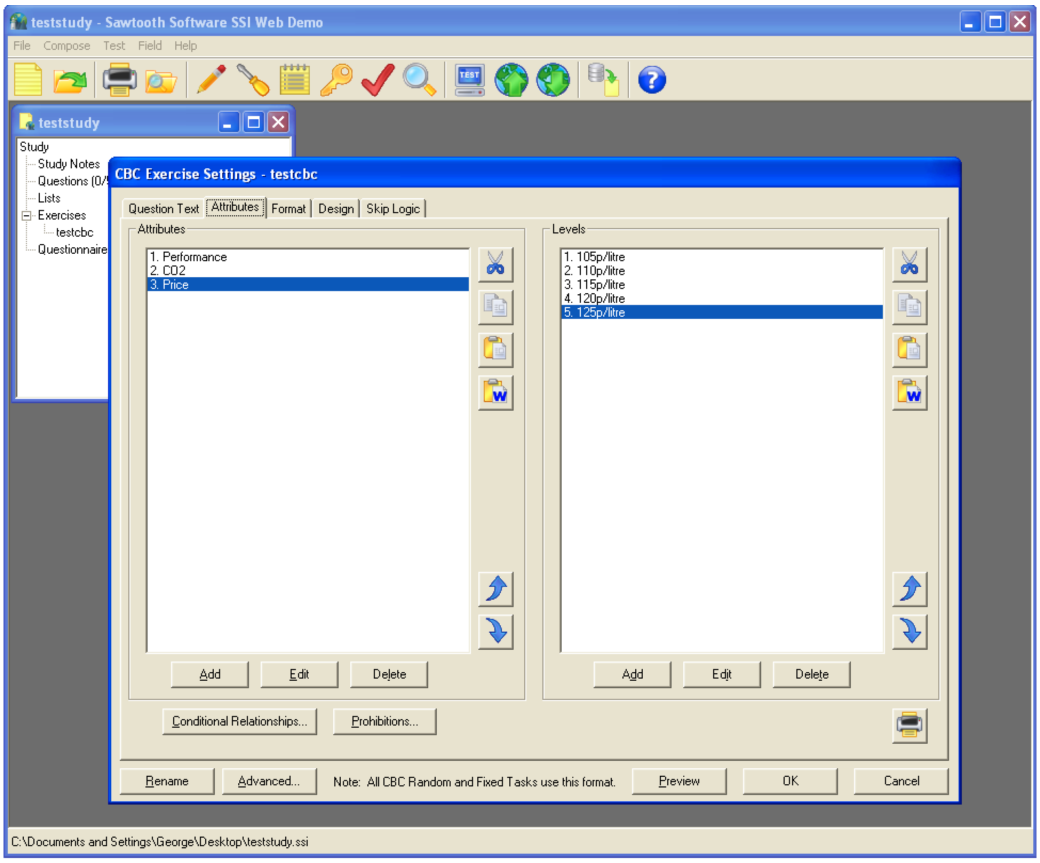
1 Have Purchased - Have you purchased an ACME product in the last year?
2 Products - Which products did you buy?
3 Dr Days - How many days did you visit your doctor last week?
4 Rating - On a scale where 5 represents "Strongly Agree" and 1 represents "Strongl
5 Gender - What is your gender?
6 Age - What is your age?

Question: 1

Type of Instruction: Skip, Delayed Skip, Avoid, Consistency Check, Randomize Answers, Invert Answers, No Duplicate Answers, Randomize Questions

Specifications: Earlier Question: Sub-question: 1; Second Earlier Q.: Sub-question: 1; Trigger Values: Formula: Trigger Sub-q: Result Question:

Figure 3: Widget overload: The Survey System software



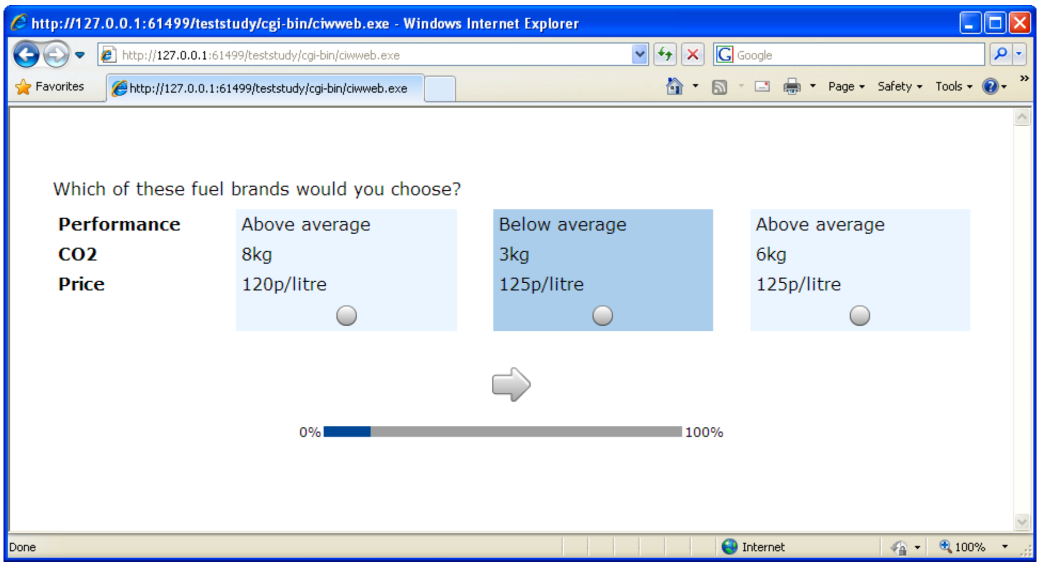
Attributes:
1. Performance
2. CO2
3. Price

Levels:
1. 105p/litre
2. 110p/litre
3. 115p/litre
4. 120p/litre
5. 125p/litre

Note: All CBC Random and Fixed Tasks use this format.

C:\Documents and Settings\George\Desktop\teststudy.ssi

Figure 4: SSI Web attribute/level specification dialogue


http://127.0.0.1:61499/teststudy/cgi-bin/ciwweb.exe - Windows Internet Explorer
http://127.0.0.1:61499/teststudy/cgi-bin/ciwweb.exe
Google
Favorites
Page
Safety
Tools
0%
100%
Done
Internet
100%

Which of these fuel brands would you choose?

Performance	Above average	Below average	Above average
CO2	8kg	3kg	6kg
Price	120p/litre	125p/litre	125p/litre
	○	○	○

Figure 5: Choice card page produced by SSI Web

(d) Do-It-Yourself development

Creating a web survey using nothing more than a general purpose programming language allows complete control and flexibility, and is an option that has been advocated elsewhere (e.g. Fraley, 2004). To do a good job, however, one needs (or needs access to) user interface expertise, visual design and programming skills, and web design experience. Even assuming all these things are available, having researchers everywhere re-implement basic web survey features—and even generic web application features—represents an extraordinary duplication of effort. It is also liable to result in little-tested and therefore buggy software, and to prove costly in time and salaries.

2.1 General limitations

None of the available options make it easy for researchers to share survey items or item types, or build on such work shared by others (in the way, say, that researchers can share their own work and make use of others' in the form of new routines or libraries for statistical software packages). Nor do these options typically make it easy to describe the underlying mechanics of a survey, either for communicating with co-authors or for reporting research methods externally (for example: What is the page-to-page skip logic? How are dynamic response options calculated? Which options are shown in randomised order?).[4]

[4]Options (a) – (c) above generally keep the workings of a survey either entirely opaque or spread throughout a series of GUI locations, while option (d) will likely produce code that is verbose and complex, where the house-keeping noise of saving data, specifying layout and so on overwhelms the details relevant to survey specification.

3 A new approach

The new approach suggested here is the use of a domain-specific language (DSL).[5] This DSL approach is related to the Do-It-Yourself development option in the discussion above, and retains its total flexibility. However, it addresses both major drawbacks of that option: first, the need for extensive expertise in web design; and second, the need to re-implement basic web application and web survey features from scratch.

3.1 A domain-specific language...

DSLs are programming languages designed to address a particular, limited problem space (as opposed to the general-purpose languages, such as C, Java, Perl and Python, in which most computer software is written). Many researchers will be familiar with one or more DSLs already. For example, in statistical packages that provide a syntax for describing and combining operations—including R, Stata, Limdep/NLOGIT, and SPSS—this syntax is well described as a DSL.

Other popular examples are LaTeX (for document typesetting/formatting), PostScript (page layout), SQL (querying and managing databases), CSS (web page formatting), Regular Expressions (text processing), XSLT (transforming XML documents) and the configuration files of the Apache web server and many other Unix tools.

3.1.1 What do DSLs do?

DSLs are designed to reflect the structure and content of the problem domain, making it straightforward to create and connect common items, actions and rules to form larger systems. As Sprinkle et al. (2009, 16) explain:

> Ideally, a DSL follows the domain abstractions and semantics as closely as possible, letting developers perceive themselves as working directly with domain concepts. The created specifications might then represent simultaneously the design, implementation, and documentation of the system...

A distinction is sometimes drawn between internal (or embedded) and external (or standalone) DSLs (Fowler, 2009). An external DSL has its own syntax, parser, and unique set of capabilities. An internal DSL, meanwhile, is a framework that extends an existing general purpose language. Some general purpose languages are more suited to hosting internal DSLs than others: key factors include the suitability of the syntax and the presence of 'meta-programming' features that make the language easy to extend.

Internal DSLs have several benefits. They are easy to implement. They get the capacities of their host language, including its existing function libraries, completely free. For those who know the host language, they use a familiar syntax; and for those that do not, they offer an easy way in to learning it.

[5] A DSL for surveys is not truly novel: there is one existing example, Topsl, described by MacHenry & Matthews (2004). Topsl's aims, capabilities and implementation are very different, however. First, Topsl requires surveys to be amenable to static analysis, so that the same source can either be displayed as web pages or formatted for printing. While this is undoubtedly useful if a hard copy is required, it removes at one stroke most of the power gained by moving surveys online. Second, Topsl is implemented in the programming language Scheme, a dialect of Lisp. Scheme is elegant, minimalist, and well suited to DSLs. However, it is also littered with parentheses and highly unlike natural language, making it rather difficult to understand (or indeed write) the survey code. Finally, Topsl includes a bare minimum of features and is not a realistic option for presenting real surveys.

3.1.2 Who are DSLs for?

DSLs are still essentially programming languages. They are picky about punctuation, capital letters and matching brackets, and getting things wrong may sometimes produce unhelpful error messages. In short, they are probably languages which those who have some basic programming experience—at the level, say, of writing an R script, Stata DO file, or SPSS syntax file—will feel more comfortable writing in.

For these developers, using a DSL may allow a substantial increase in productivity. In general, DSLs can cut development time by between 60% and 90% compared to standard, general-purpose programming approaches (Kelly & Tolvanen, 2008, 22–25). The value of using a DSL is by no means limited to those who write in it, though. As (Fowler, 2009, 14) notes:

> the key value is providing a business-readable DSL, where domain experts can read the code, understand what it means, and talk to programmers directly about necessary modifications. It's much easier to make DSLs business readable rather than business writable, but you gain most of the benefits by enhancing communication.

3.1.3 Introducing *websperiment*

websperiment is a prototype of an internal DSL for specifying web surveys and experiments. Its host language is Ruby, a dynamic, interpreted language (Flanagan & Matsumoto, 2008, 2). Ruby's syntax is relatively close to that of natural language, and it has strong facilities for 'meta-programming', a means of extending the language by using code that itself writes code. These attributes make it well suited to creating internal DSLs (in fact, many current Ruby users were introduced to the language by a popular DSL for writing generic web applications, Rails[6]).

As with other DSLs, the intention of *websperiment* is that domain experts should be able to understand a survey that is implemented in it just by reading the code, even if they do not initially feel confident writing this code themselves. With that hope in mind, a very simple first example is presented: a survey with two pages and two questions. The DSL code is shown in Listing 1, and the resulting survey in Figure 6.

In Listing 1, line 1 creates a new survey. By default the survey's title, which is shown at the top of the browser window and at the top of the web page, is taken from the name given on this line (here, for example `S::ExampleSurvey` becomes "Example survey"). The content of the survey is defined by the block, surrounded by `do ... end` markers, that starts on this line and comprises the rest of the listing. Within that block, line 2 introduces the pages of the survey, whose definitions then start on lines 3 and 16. As is the case for the survey as a whole, the content of each page is defined in the `do ... end` block immediately following its declaration. The first page contains some text, followed by two questions (again defined by the `do ... end` blocks that follow them). The second page contains only some text, and a declaration that this page completes the survey.

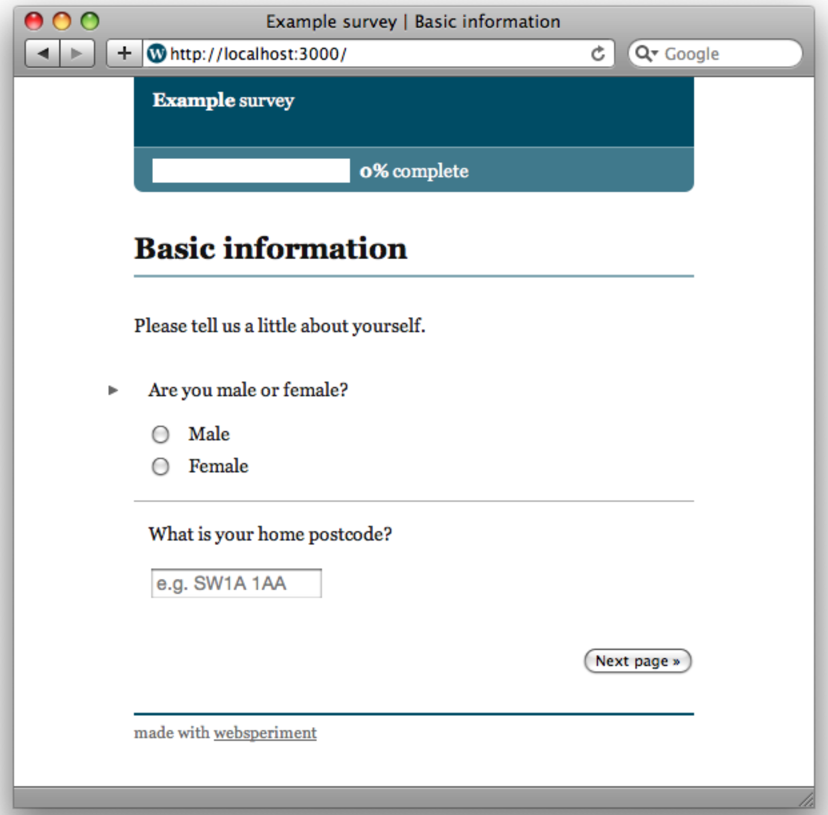
Admittedly, this is not a very interesting survey. In fact, it stays well within what is possible using a free web-based service. However, the key advantage of the DSL approach is that it allows for this near-effortless implementation of standard survey features, without constraining the researcher's freedom to do essentially anything at all.

The example survey page in Listing 2 and Figure 7 gives a simple illustration of both of these aspects. It specifies how to deal with blank responses using a simple declaration (lines 12 – 13). It also uses the web service API of the They Work For You (TWFY) web site[7] to look up a

[6] `http://rubyonrails.org/`.
[7] See `http://theyworkforyou.com` and `http://github.com/bruce/twfy`.

```
S::ExampleSurvey = S::Base.declare_new do
  pages(
    P::BasicInformation = P::Base.declare_new do
      text "Please tell us a little about yourself."
      questions(
        Q::Male = Q::Radio.declare_new do
          text "Are you male or female?"
          options [1, "Male"],
                  [0, "Female"]
        end,
        Q::HomePostcode = Q::Postcode.declare_new do
          text "What is your home postcode?"
        end
      )
    end,
    P::ThankYou = P::Base.declare_new do
      text "Many thanks for completing this survey."
      completes_survey true
    end
  )
end
```

Listing 1: *websperiment* code specifying a simple web survey.



Basic information

Please tell us a little about yourself.

Are you male or female?

Male

Female

What is your home postcode?

e.g. SW1A 1AA

Figure 6: First page of the survey specified in Listing 1.

```
1  P::EnvironmentalConcern = P::Base.declare_new do
2    questions(
3      Q::ContactedMP = Q::Radio.declare_new do
4        require "twfy"
5        mp = Twfy::Client.new("MY_API_KEY").mp(postcode: Q::HomePostcode.answer) rescue nil
6        text "*#{mp.full_name}* -- #{mp.party} MP for #{mp.constituency.name} -- is your MP.",
7             style: :info unless mp.nil?
8        text "Have you ever contacted your MP about an environmental issue?"
9        options [1, "Yes"],
10               [0, "No"],
11               [99, "Not sure"]
12       completion :prompted,
13                  message: "*Did you miss this question?* You don't have to answer, but knowing
       whether you have contacted your MP would be very useful to our research."
14     end
15   )
16 end
```

Listing 2: Page with a dynamic question, using a previous answer and an external web service.

UK respondent's local MP based on a postcode entered on the previous page (lines 4 – 5) and display this as part of the question text (lines 6 – 7).[8]

3.2 ...with inheritance...

websperiment gains some of its most useful capabilities from an approach known as object-oriented programming (OOP). OOP has its roots in the 1960s, and has been mainstream in software development for several decades.

The objects of OOP are conceptually cohesive entities that generally model and reflect things in the outside world. Objects consist both of data and of behaviours (or 'methods') that work with that data. Objects interact by passing messages, which ask that specific methods be invoked. OOP uses these objects, and their interactions, in the design and implementation of larger systems (Snyder, 1986; Armstrong, 2006). For example, OOP is increasingly used in agent-based modelling (Benenson & Torrens, 2004): objects in these studies can model agents such as the vehicles within traffic flows, or households making decisions regarding where to locate within a region (Torrens & Nara, 2007).

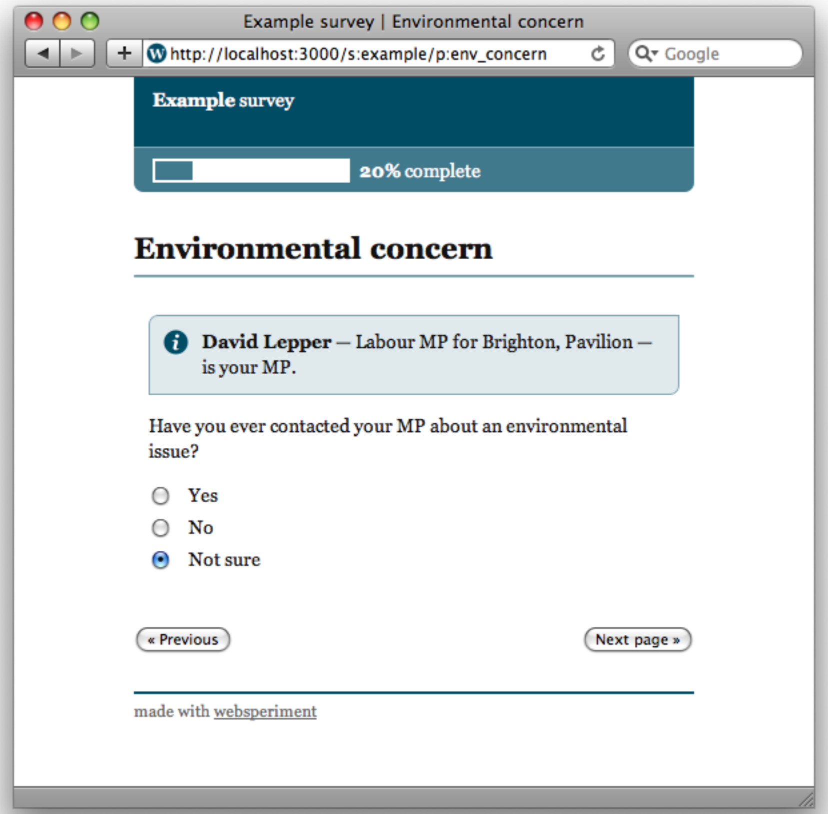
OOP systems commonly make a distinction between a *class*, which is the definition or 'blueprint' for an object, and an *instance*, which is a specific realisation of its class. However, since almost every object in *websperiment* is a *singleton* object — one which is intended only ever to have a single instance — this distinction not important in the use of the DSL.

websperiment is built using three main families of objects. At the core are question objects (these have names that start with `Q::`). For display, questions are composited into page objects (starting with `P::`), and for navigation from page to page, pages are composited into Survey objects (`S::`).

Objects in an OOP system can 'inherit' data and behaviours from other objects in a hierarchy or tree. This makes it easy to create objects that build on or modify the abilities of other objects. In OOP terminology, one creates subclasses of (or one simply 'subclasses') the original object class.

This process was seen in action in Listing 1. When the first survey page was created, it was declared as a new subclass of the basic page class, `P::Base` (by the line `P::BasicInformation =`

[8]In Ruby, and therefore *websperiment*, `#{}` does string interpolation: code placed between the curly braces within a string of text is executed and substituted into that text.



Example survey | Environmental concern

http://localhost:3000/s:example/p:env_concern

Example survey

20% complete

Environmental concern

David Lepper — Labour MP for Brighton, Pavilion — is your MP.

Have you ever contacted your MP about an environmental issue?

- Yes
- No
- [✓ Not sure]

« Previous

Next page »

made with websperiment

Figure 7: Survey page specified in Listing 2.

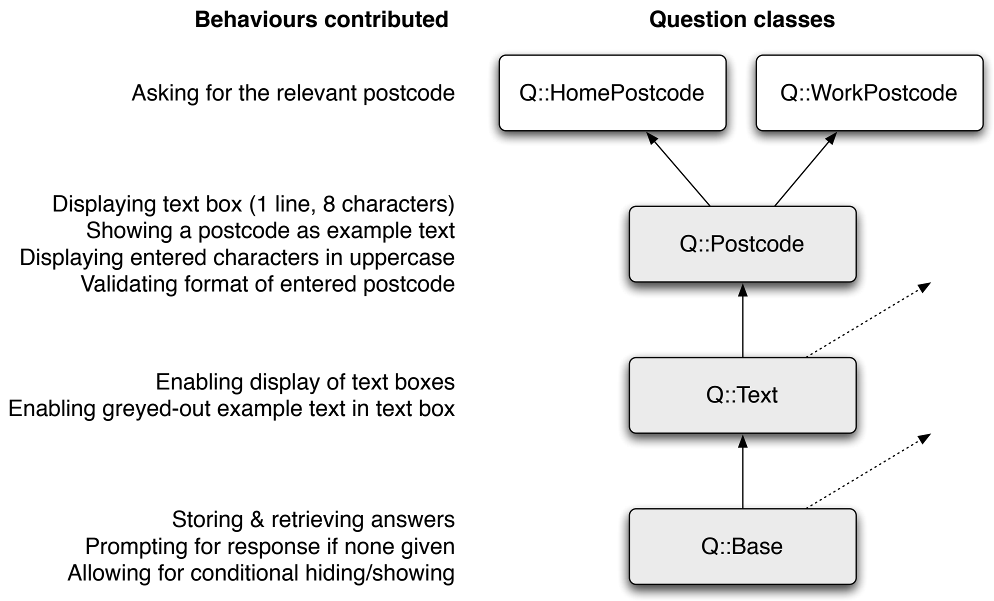

Behaviours contributed
Question classes
Asking for the relevant postcode
Q::HomePostcode
Q::WorkPostcode
Displaying text box (1 line, 8 characters)
Showing a postcode as example text
Displaying entered characters in uppercase
Validating format of entered postcode
Q::Postcode
Enabling display of text boxes
Enabling greyed-out example text in text box
Q::Text
Storing & retrieving answers
Prompting for response if none given
Allowing for conditional hiding/showing
Q::Base

Figure 8: Inheritance hierarchy for postcode questions.

`P::Base.declare_new`). The new page therefore inherited all the standard data and behaviours required of a survey page: generating an HTML form, displaying progress through the survey as a progress bar, and so on. A block of code was then added, between the `do ... end` markers, to augment this with custom data and behaviours (in this case, some text and some questions).

Similarly, the gender question, `Q::Male`, was created as a new subclass of a radio-button question class, `Q::Radio`. It thus inherited all the data and behaviours required of a radio-button question: specifying response options, displaying these as radio buttons, and subsequently processing the respondent's selection. Again, additional data and behaviours were then specified in the subsequent block of code (in this case, some question text and two response options).

There is in fact no conceptual distinction between question classes that are built in to *websperiment* and those that are created by a researcher: all built-in classes are created in exactly the same way that new custom classes are. For example, the `Q::Postcode` class used in Listing 1 inherits from the `Q::Text` class, which itself inherits from a basic question class, `Q::Base`. At each level of the inheritance hierarchy, new data and behaviours build on those already defined. This hierarchy for `Q::Postcode`, and some examples of the behaviours defined at each level, are illustrated in Figure 8.

3.2.1 Templating with abstract classes

Subclassing is not limited only to the built-in question, page and survey classes. Say, for example, that a researcher wants to ask respondents a sequence of several yes/no questions, and include an indication of their certainty about each response. The researcher could create new intermediate question subclasses for this purpose (they might be named `Q::YesNo` and `Q::Certainty`). These intermediate subclasses are never themselves displayed, and do not appear when the response data is downloaded from the survey, because they are never directly added to any page. In OOP terminology they are known as *abstract* classes, and in *websperiment* they act as templates. Within the pages of a survey, these abstract classes can be subclassed as necessary.

Listing 3 and Figure 9 illustrate this. The `insert_future_layout` declaration (line 2) specifies where the content contributed by any descendant question classes should be inserted. The horizontal scale question class subclassed by the certainty class (line 7) provides a new kind of

```
 1  Q::YesNo = Q::Radio.declare_new do
 2    insert_future_layout
 3    options [1, "Yes"],
 4            [0, "No"]
 5  end
 6
 7  Q::Certainty = Q::ScaleHorizontal.declare_new do
 8    text "How certain are you of this?"
 9    scale range: 1..5,
10          start: "Uncertain",
11          end:   "Certain"
12  end
13
14  P::PurchasingDecisions = P::Base.declare_new do
15    questions(
16      Q::FivePounds = Q::YesNo.declare_new do
17        text "Would you buy item X for £5?"
18      end,
19      Q::FivePoundsCertainty = Q::Certainty.declare_new,
20      Q::TenPounds = Q::YesNo.declare_new do
21        text "Would you buy item X for £10?"
22      end,
23      Q::TenPoundsCertainty = Q::Certainty.declare_new
24    )
25  end
```

Listing 3: Simple inheritance example.

declaration, `scale`, which is used to define the placement of the response scale and its numeric range and labels (lines 9 – 11).

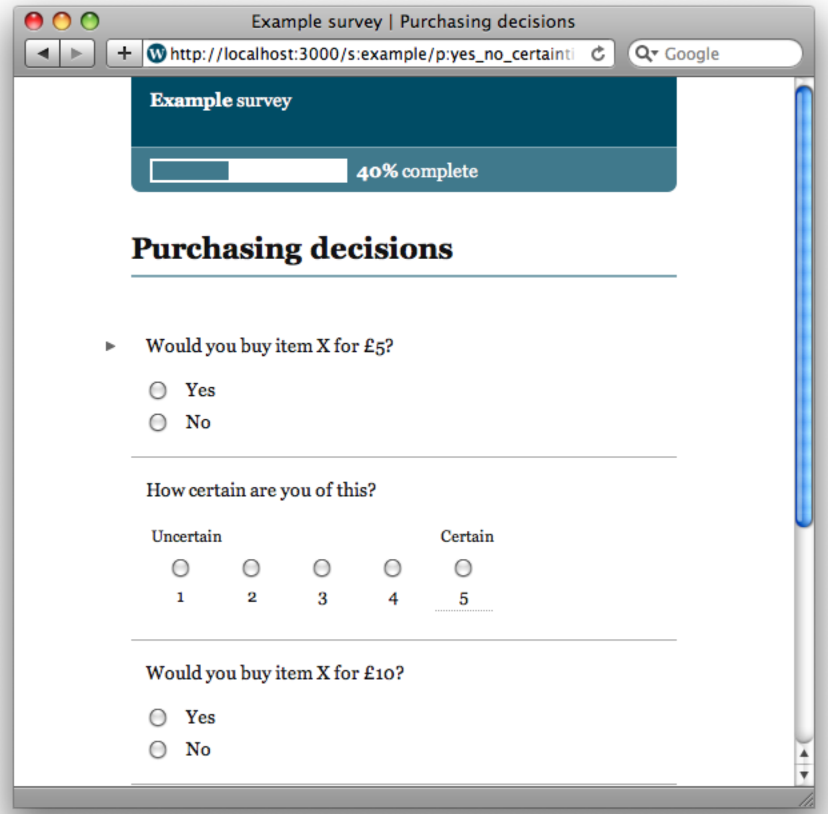
If the researcher later decides to switch from a 1 – 5 scale to a 0 – 100% scale, all he or she has to do is modify the `Q::Certainty` class, and the change is automatically reflected in all the questions which subclass and thus inherit from it. This helps to eliminate repetition, reduce inconsistencies, and improve the maintainability of the survey code. The `Q::Certainty` subclass could be reused elsewhere in the same survey, and in any of the researcher's future surveys. It could also be shared with other researchers, for use in their surveys.

The example given above is short and simple enough that the advantages of these possibilities are not very great. However, question, page and survey classes need not be limited to a few simple lines of layout. As seen in the next subsection, they may include complex customised styling, content and logic, on both the server and the client (the web browser).

3.2.2 Advanced question types

Question types may be designed to collect survey paradata. Some of these are datum (`Datum::`) classes. These are similar to question (`Q::`) classes, in that they record an item of information for each respondent; however, they do not cause anything to be shown to the respondent (or even to be sent to their web browser). For example, the following are all available within *websperiment*:

- a `Datum::TimeViewed` class which, rather than asking the respondent any question, records the time he/she arrives on a page (which can be used to calculate the time spent on each page);
- a `Q::AnswerHistory` class, which automatically monitors another question and records the sequence of answers selected;


Example survey | Purchasing decisions

http://localhost:3000/s:example/p:yes_no_certainti

Example survey

40% complete

Purchasing decisions

Would you buy item X for £5?

- Yes
- No

How certain are you of this?

Uncertain				Certain
1	2	3	4	5

Would you buy item X for £10?

- Yes
- No

Figure 9: Survey page specified in Listing 3.

```
1  P::MapLocation = P::Base.declare_new do
2    questions(
3      Q::HomeMapLocation = Q::MapLocation.declare_new do
4        depends_on Q::HomePostcode
5        text "*Please click and drag* to move the map so that your home is inside the yellow box."
6        map start_location: Q::HomePostcode.answer,
7            size:           "390x300"
8      end
9    )
10 end
```

Listing 4: Using the map location question class.

- a `Q::LinkVisited` class, which interrogates the respondent's web browser history to determine whether he/she has recently visited a specific web address[9]; and
- a `Datum::CityFromIPAddress` class, which uses public geo-location data to determine where a respondent is located.

Wholly new interactive question types can also be developed, such as visual analog scales (Couper et al., 2006) or interactive maps. For a recent survey in which precise geographical location was an important variable a `Q::MapLocation` class was created. This asks the respondent to pinpoint a precise location by dragging a satellite mapping image, and records the latitude and longitude of that location, as shown in use in Listing 4 and Figure 10.

In Listing 4, the map location question provides a new type of declaration, `map` (lines 6 – 7), which specifies where the map is to be displayed within the question content, its size in pixels, and its starting location, the latter based in this case on the answer to a previous question. The map question is also declared as dependent on the postcode question whose answer it uses to centre the map initially displayed (line 4). The effect of this is to clear the location recorded if the respondent goes back and changes the postcode entered, with the effect that the map is re-centred on the newly entered postcode.

3.3 ...that is open source

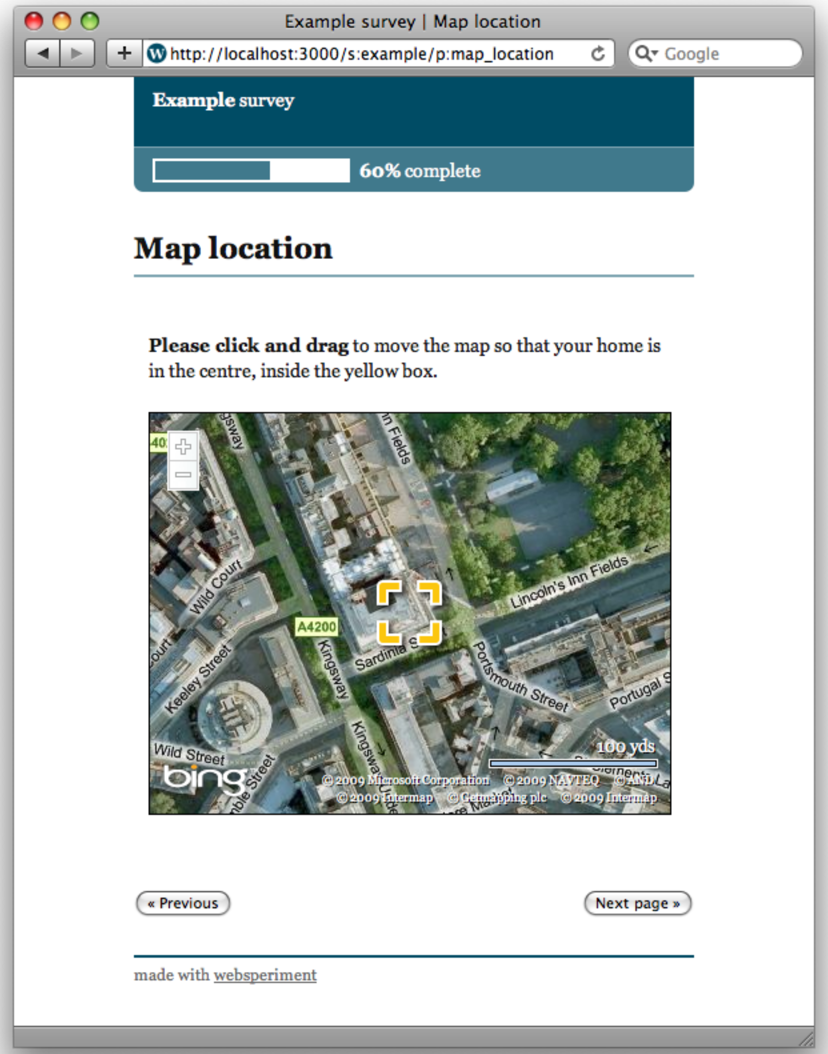
Of course, not every researcher will have the expertise in Ruby (for the server) or HTML, JavaScript, and CSS (for the client) to create advanced subclasses of this kind. However, it is easy to use subclasses created and shared by others. Anyone can use `Q::MapLocation` in the way shown in Listing 4 simply by downloading the code for the class and adding it to their *websperiment* library. This is very similar to the way in which capabilities can be added to R or Stata by downloading modules or packages created by others. For the survey realm, it seems somewhat overdue.[10]

To facilitate such sharing, *websperiment* is built wholly from open-source parts, and its own code is also released under an open-source licence. This means that researchers are free to download, modify, run and distribute the code. This makes *websperiment* particularly compatible with an academic ethos and the scientific method.[11] Collaborative working, transparency, peer review, and reproducibility are all made easier to achieve, since surveys (as well as the whole DSL framework) can be easily shared, evaluated, modified, improved, and re-run.

[9]Obviously a respondent's informed consent must be sought before deploying a survey item of this type.

[10]Of course, there is no obligation to share code, and a class that reveals too much about confidential research might be unsuitable for sharing. Additionally, to guard against the proliferation of poorly-written or excessively similar classes, some form of curated repository might turn out to be of benefit in the longer term.

[11]For further discussion of the relationship between academia and open source software, see Lerner & Tirole (2005).



Example survey | Map location

http://localhost:3000/s:example/p:map_location

Example survey

60% complete

Map location

Please click and drag to move the map so that your home is in the centre, inside the yellow box.

made with websperiment

Figure 10: Survey page specified in Listing 4.

3.4 Technical notes

Although defaults have been chosen with care, the presentation of surveys is entirely customisable. Ease of customisation depends, obviously, on the extent of the customisations desired. Images are easily included within question and page declarations. Colours, fonts, and dimensions can be altered with simple modification to CSS stylesheets. Larger structural changes might require modification of HTML templates, or tweaks to the Ruby code.

On the respondent side, surveys are marked up in standards-compliant HTML, CSS and JavaScript, and are compatible with all major browser platforms, including Internet Explorer 6+, Mozilla (including Firefox), Webkit (including Safari, Chrome, the iPhone's Mobile Safari, and the Android Browser) and Opera. All built-in JavaScript is unobtrusive, in the sense that users without it are still able to complete the survey, although questions that rely on JavaScript may be unavailable, and there may be intra-page routing instructions to follow.

On the server side, *websperiment* requires Ruby 1.9 and various Ruby *gems* (libraries), including Rails 2.3. Like any Rails application, it can be hosted by various web server packages—Apache or Nginx, with the Phusion Passenger module, are recommended—and it works with major databases including MySQL, PostgreSQL, Oracle and SQL Server. Someone comfortable working at the command line could expect to get these different elements of the system installed and running in well under an hour.

4 Extending *websperiment* for choice experiments

A surveys DSL is well suited to constructing web-based choice experiments, since these can be relatively complex and dynamic surveys. For a recent study on vehicle fuel choice, several reusable custom object classes were developed within *websperiment*. These classes are now shared as part of the *websperiment* library for other researchers to use (though the creation of such classes from scratch is not necessarily a major task: those discussed below represent no more than than a day's work). This section discusses a bare-bones choice experiment based on these classes.

The experiment aims to value CO_2 emissions and engine performance characteristics of vehicle fuels. It retrieves UK fuel price data in real-time, dynamically setting the 'status quo' fuel price as the current average fuel price within 10 miles of the respondent's postcode,[12] and all other fuel price attribute levels as deviations from this value. It assembles choice cards (see Figure 11) out of options that are random combinations of attribute levels, eliminating cards where any option is duplicated, or dominated by any other. Finally, it routes respondents who choose the status quo option every time to a special debriefing page (pictured in Figure 12).

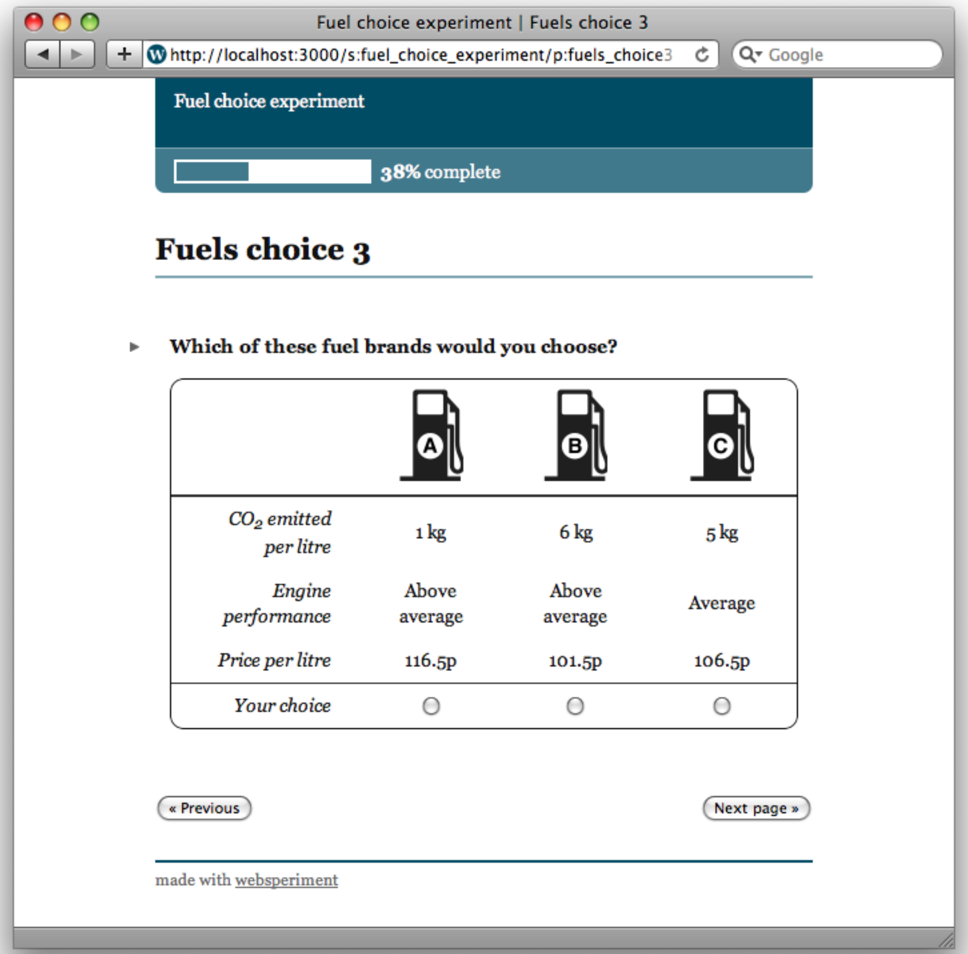
The experiment is viewable online at `http://choicesurvey.websperiment.org/`. Its DSL source is under 100 lines of code, described further below and shown in full in Listings 5 – 9. Note that it is *not* intended to represent good survey design or good experimental design, but only to demonstrate the potential benefits of a DSL in this context.

4.1 Getting a local fuel price

The first page of the survey simply requests the respondent's postcode, as in the first example (subsubsection 3.1.3). This postcode is then used by a simple custom `Datum::` class to retrieve the average local fuel price (because it is a `Datum::` class, it stores a value alongside the respondent's answers, but does not display anything to the respondent).

These steps are seen in Listing 5. On line 9, the status quo price item is defined as a subclass of the local fuel price item. On line 10, it is made dependent on the postcode question (so that

[12]According to `http://petrolprices.com/`.



Fuel choice experiment | Fuels choice 3

http://localhost:3000/s:fuel_choice_experiment/p:fuels_choice3

Fuel choice experiment

38% complete

Fuels choice 3

Which of these fuel brands would you choose?

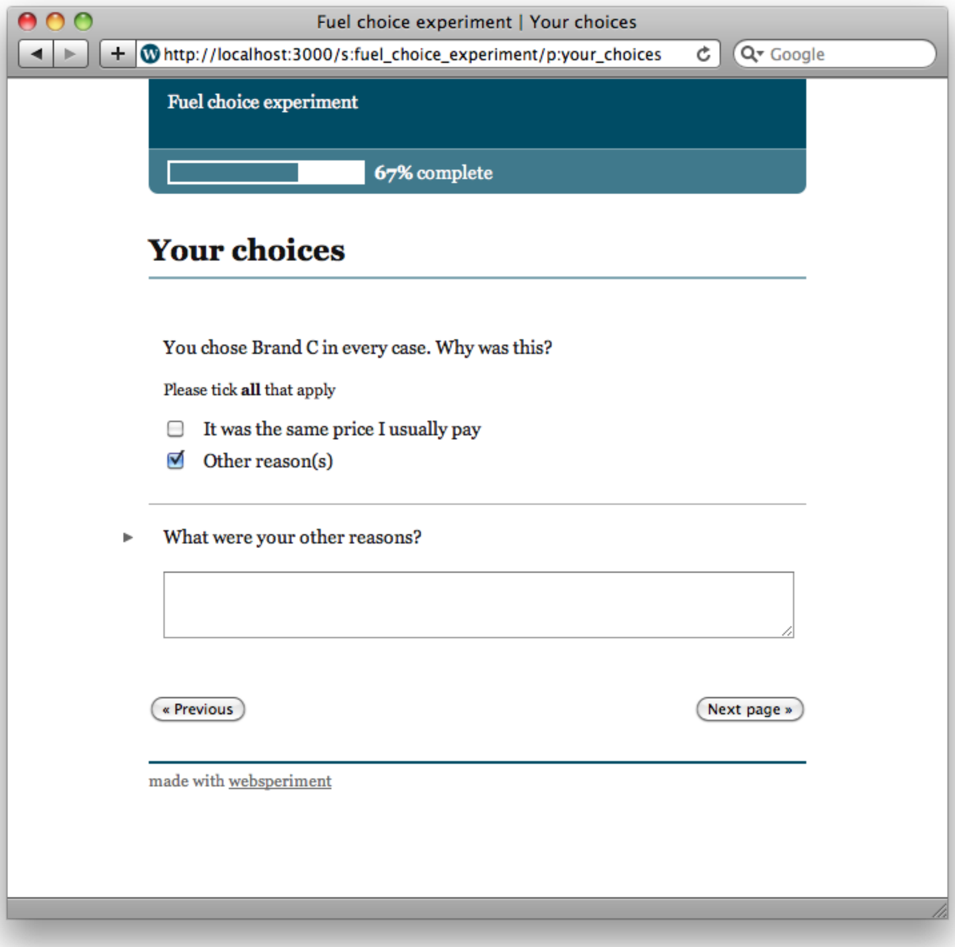
	A	B	C
CO_2 emitted per litre	1 kg	6 kg	5 kg
Engine performance	Above average	Above average	Average
Price per litre	116.5p	101.5p	106.5p
Your choice	○	○	○

« Previous

Next page »

made with websperiment

Figure 11: Survey page showing randomised choice card with dynamic price attribute levels.


Fuel choice experiment | Your choices
http://localhost:3000/s:fuel_choice_experiment/p:your_choices
Google
Fuel choice experiment
67% complete

Your choices

You chose Brand C in every case. Why was this?

Please tick **all** that apply

- [ ] It was the same price I usually pay
- [x] Other reason(s)

What were your other reasons?

« Previous
Next page »
made with websperiment

Figure 12: Follow-up survey page to be shown based on choice pattern.

```
 1  P::HomePostcode = P::Base.declare_new do
 2    questions(
 3      Q::HomePostcode = Q::Postcode.declare_new do
 4        text "What is your home postcode?"
 5      end
 6    )
 7  end
 8
 9  Datum::StatusQuoPrice = Datum::LocalFuelPrice.declare_new do
10    depends_on Q::HomePostcode
11    get_price for_postcode:   Q::HomePostcode.answer,
12              fallback_price: 109
13  end
```

Listing 5: Determining the average fuel price in the respondent's local area.

```
 1  Datum::FuelSet = Datum::AttributeSet.declare_new do
 2    depends_on Datum::StatusQuoPrice
 3    attribute :co2_emitted,
 4              levels:     1..9,
 5              status_quo: 5,
 6              more_is:    :worse
 7    attribute :engine_performance,
 8              levels:     0..2,
 9              status_quo: 1,
10              more_is:    :better
11    attribute :price,
12              levels:     [-10, -5, 0, 5, 10].map { |d| d + Datum::StatusQuoPrice.value },
13              status_quo: Datum::StatusQuoPrice.value,
14              more_is:    :worse
15  end
16
17  Datum::ThreeWayFuelSet = Datum::FuelSet.declare_new do
18    random_set labels:           ["a", "b"],
19               status_quo_label: "c"
20  end
```

Listing 6: Defining choice attributes and levels.

its value will be cleared and recalculated whenever the answer to that question changes). On line 11, it is set to retrieve the price for the postcode provided by the respondent. On line 12, a fallback price is specified: this fallback price will be used if no postcode is provided, or if the postcode is not recognised, or if the request to petrolprices.com fails to return an answer within an acceptable timeout period.

4.2 Defining attributes

The next step in creating the choice experiment is defining the attributes and levels. This is accomplished using another reusable custom `Datum::` class, as seen in Listing 6.

The first attribute is defined on lines 3 – 6. Line 3 gives the attribute a name, by which it can be identified elsewhere in the code. Line 4 defines its levels: the series of integers from 1 to 9 inclusive. Line 5 specifies the status quo level, which will be used as the value for the status quo option to be shown on every choice card. Line 6 specifies that higher values are less desirable: this information enables the randomisation algorithm to detect when one option

```
1  Q::FuelsCard = Q::ChoiceCard.declare_new do
2    text "*Which of these fuel brands would you choose?*"
3    label_format -> l { "!/images/pump_#{l}.png(Brand #{l.upcase})!" }
4    display :co2_emitted,
5            title:  "CO[~2~] emitted per litre",
6            format: -> c { "#{c} kg" }
7    display :engine_performance,
8            title:  "Engine performance",
9            format: -> ep { ["Below average", "Average", "Above average"][ep] }
10   display :price,
11           title:  "Price per litre",
12           format: -> p { "#{p}p" }
13 end
14 
15 P::FuelsChoice = P::Base.declare_new do
16   choice_attributes = Datum::ThreeWayFuelSet.declare_new.named_with_suffix(name_suffix)
17   datum choice_attributes
18   questions(
19       Q::FuelsCard.declare_new do
20         depends_on choice_attributes
21         attributes choice_attributes.value
22       end.named_with_suffix(name_suffix)
23   )
24 end
```

Listing 7: Formatting choice cards.

(i.e. a combination of levels) is dominated by another.[13] For qualitative attributes, or other attributes where the desirability of different levels is not known in advance, this line would be omitted.

The second attribute is defined similarly on lines 7 – 10, and the third on lines 11 – 14. For the third attribute, the status quo level is set dynamically to the respondent's local fuel price as already queried online (line 13). The full set of levels is then defined as deviations from this value (line 12).[14] For this reason, the whole attribute set is specified as dependent on the status quo price (line 2). This means that if there is a change in the status quo price (which could in this case be caused only by a change to the postcode), the attribute levels will be cleared and recalculated accordingly.

Then, having defined the attributes and their levels, attributes must be combined into options, and options combined into choice cards. This is accomplished by lines 17 – 20, which specify that two randomised options are to be generated, labelled 'a' and 'b', alongside a status quo option, labelled 'c'.

4.3 Displaying choice cards

Once created, the choice cards must be displayed to respondents (as illustrated in Figure 11). This is achieved by a custom question (`Q::`) class, as seen in Listing 7. This class allows formatting to be specified for each attribute, and then displays HTML tables based on specific sets of attribute values as generated in Listing 6.

Lines 1 – 13 of Listing 7 define the formatting to be used for the choice card. On line 3, the option labels ('a', 'b', and 'c') are used as parts of the file names of images which are used as

[13]Option A is dominated by option B if it is worse than B in one or more of its attributes, and not better in any.

[14]The `map` method on line 12 maps the array of values on its left to a new array of values, by passing each original value into the block that follows, as the value of the variable `d`, and using the values returned by the block as the values comprising the new array.

```
1  P::YourChoices = P::Base.declare_new do
2    questions(
3      Q::StatusQuoAlwaysReason = Q::Checkbox.declare_new do
4        text "You chose Brand C in every case. Why was this?"
5        text "Please tick *all* that apply", style: :smaller
6        options [:price, "It was the same price I usually pay"],
7                [:other, "Other reason(s)"]
8      end,
9      Q::StatusQuoAlwaysOther = Q::Text.declare_new do
10       show_if "Q::StatusQuoAlwaysReason.answer.include? :other",
11               human: "you chose 'Other reason(s)' above"
12       text "What were your other reasons?"
13       text_box size:       "70x3",
14                full_width: true
15     end
16   )
17 end
```

Listing 8: Debriefing page for a specific choice pattern, as shown in Figure 12.

the table headings. Lines 4 – 12 define the attribute titles and formatting (on line 9, the engine performance attribute is used as an index to an array of text strings: thus, when the attribute level is 0, 'Below average' will be displayed; when it is 1, 'Average' will be displayed; and so on).

Lines 15 – 23 define the choice card survey page. Line 16 creates a new set of attribute levels (as a subclass of the second `Datum::` class defined in Listing 6). The new class is named not by assignment—the way that has been seen before (e.g. `Datum::X = Datum::Y.declare_new`)—but using the method `named_with_suffix`, which simply appends a number or text string as a suffix to the name of the class that is being subclassed. The argument passed to this method here—`name_suffix`—returns the suffix part of the page's own name (since, as will be seen in Listing 9, the choice card survey pages are themselves named using the `named_with_suffix` method). The new class is also assigned to a local variable, `choice_attributes`, for later use.

Line 17 adds the newly created set of attribute levels to the page (and hence to the survey: without this line, the levels would not be stored). Finally, the choice card question is added to the page on lines 18 – 23. Line 21 specifies that the question should display the attribute levels created on line 16. Line 20 specifies the question's dependency on these levels: if the levels change (due to change in the status quo price, due to a change in the postcode), the answer entered for this question will be cleared. On line 22 the choice card question is named using `named_with_suffix`.

The choice card page and question classes are named using suffixes because they will be subclassed, and added to the survey, multiple times—once each per choice card—but still require unique names by which they may be identified (both elsewhere in the survey and when saving and reporting the respondents' answers).

4.4 Following up response patterns

Once the respondent has made the choices presented, it may be valuable to ask follow-up debriefing questions in response to specific choice patterns. In this example, respondents who always choose option C (the status quo option) are asked their reasons for doing so. This is illustrated in Figure 12, and the DSL code is shown in Listing 8.

Of greatest interest here are probably lines 10 – 11. These lines specify that the second question is to be displayed only where the respondent ticks the 'Other reason(s)' check-box on the

```
1  P::Thanks = P::Base.declare_new do
2    text "*Thank you* for completing this survey"
3    completes_survey true
4  end
5
6  S::FuelChoiceExperiment = S::Base.declare_new do
7    n = 3
8    page P::HomePostcode
9    (1..n).each { |card_no| page P::FuelsChoice.declare_new.named_with_suffix(card_no) }
10   pages(
11     P::YourChoices, skip_unless { (1..n).all? { |card_no| eval("Q::FuelsCard#{card_no}").answer == 'c' } },
12     P::Thanks
13   )
14 end
```

Listing 9: Pulling the experiment together.

first question. Line 10 gives the condition in *websperiment* (Ruby) format (this condition is also translated into JavaScript, for use by the respondent's web browser[15]). Line 11 gives the condition as it is to be displayed to the respondent, if JavaScript is for any reason unavailable in the browser.

4.5 Putting the experiment together

Finally, the components seen above need to be integrated into a survey, as shown in Listing 9. Lines 1 – 4 define an acknowledgement page. On line 2, note the use of asterisks (*) around some of the text. This is one example of the Textile markup system used by *websperiment*: it causes the text to be displayed in **boldface**.[16]

Lines 6 – 14 define the overall survey object, bringing together the previously-defined page objects. Line 7 sets the number of choice cards to be displayed, as a local variable `n`. Line 8 adds the postcode question page to the survey. On line 9 all the choice choice card pages are added, as follows: the integers 1 to `n` are passed into the block as the value of `card_no`, and a new subclass of the `P::FuelsChoice` page is created each time, named with that integer as a suffix (`P::FuelsChoice1`, `P::FuelsChoice2`, and so on), and added to the survey by the `page` declaration. Lines 10 – 13 add the follow-up debriefing page and the acknowledgement page, and line 11 also defines the condition according to which the debriefing page will be displayed (it will be skipped *unless* option 'c' was selected on *all* of the choice cards).

5 Conclusions

Researchers have up to now usually had to pick one of four main options when implementing surveys and choice experiments online: web-based services, local software, consultants, or Do-It-Yourself from scratch.

Implementing a web-based survey or experiment using a DSL such as the *websperiment* prototype, it has been argued, has a number of advantages over the options previously available. These include: improved productivity, leading to reduced time and cost of implementation; greater flexibility and extensibility, enabling arbitrarily advanced features; and wider accessibility and browser compatibility, permitting a broader sampling frame and a superior experience for respondents.

[15]The 'translation' is accomplished using an enhanced version of a system called HotRuby: see `http://hotruby.yukoba.jp/` and `http://github.com/STRd6/hotruby`.

[16]For further details, see `http://redcloth.org/textile`.

The advantages also include easier communication and sharing, both within a specific research project and (where relevant) more broadly within the academic process. This is for three main reasons: first, because the DSL is readable by domain experts who need not be experts in web survey implementation; second, because it is designed in a modular way, using reusable, sharable classes; and third, because it is unencumbered by proprietary licence restrictions.

websperiment has already been used to implement several online surveys and experiments, but it is not a finished product. Researchers are strongly encouraged to use and contribute to the development of the project.

References

Armstrong, D. J. (2006). The quarks of object-oriented development. *Communications of the ACM*, 49(2), 123–128.

Benenson, I. & Torrens, P. M. (2004). Geosimulation: object-based modeling of urban phenomena. *Computers, Environment and Urban Systems*, 28(1-2), 1–8.

Couper, M. P. (2008). *Designing effective web surveys*. Cambridge University Press.

Couper, M. P., Tourangeau, R., Conrad, F. G., & Singer, E. (2006). Evaluating the effectiveness of visual analog scales. *Soc. Sci. Comput. Rev.*, 24(2), 227–245.

Dillman, D. A., Tortora, R. D., & Bowker, D. (1998). *Principles for Constructing Web Surveys*. SESRC Technical Report 98-50, Pullman, Washington.

Flanagan, D. & Matsumoto, Y. (2008). *The Ruby Programming Language*. O'Reilly Media, Inc.

Fowler, M. (2009). A pedagogical framework for Domain-Specific Languages. *Software, IEEE*, 26(4), 13–14.

Fraley, R. C. (2004). *How to Conduct Behavioral Research over the Internet: A Beginner's Guide to HTML and CGI/Perl*. New York: Guilford Press.

Heerwegh, D. (2003). Explaining response latencies and changing answers using client-side paradata from a web survey. *Social Science Computer Review*, 21(3), 360–373.

Kelly, S. & Tolvanen, J. (2008). *Domain-specific modeling: enabling full code generation*. Wiley-IEEE.

Kuipers, M. (2005). Review of web-based survey tools. UCL Information Systems resource.

Lerner, J. & Tirole, J. (2005). The economics of technology sharing: Open source and beyond. *The Journal of Economic Perspectives*, 19(2), 99–120.

MacHenry, M. & Matthews, J. (2004). Topsl: A domain-specific language for on-line surveys. Fifth Workshop on Scheme and Functional Programming.

Marta-Pedroso, C., Freitas, H., & Domingos, T. (2007). Testing for the survey mode effect on contingent valuation data quality: A case study of web based versus in-person interviews. *Ecological Economics*, 62(3-4), 388–398.

Schonlau, M., Fricker, R. D., & Elliott, M. N. (2002). *Conducting Research Surveys via E-mail and the Web*. RAND research.

Snyder, A. (1986). Encapsulation and inheritance in object-oriented programming languages. *ACM Sigplan Notices*, 21(11), 45.

Sprinkle, J., Mernik, M., Tolvanen, J., & Spinellis, D. (2009). Guest editors' introduction: What kinds of nails need a Domain-Specific hammer? *Software, IEEE*, 26(4), 15–18.

Torrens, P. M. & Nara, A. (2007). Modeling gentrification dynamics: A hybrid approach. *Computers, Environment and Urban Systems*, 31(3), 337–361.